Some Potential Issues with the Security of HTML5 IndexedDB

Stefan Kimak , Jeremy Ellman, Christopher Laing
Northumbria University, Faculty of Engineering and Environment
Newcastle upon Tyne, UK
stefan.kimak@ northumbria.ac.uk , jeremy.ellman@ northumbria.ac.uk, christopher.laing@ northumbria.ac.uk

Keywords*:* Component web security; IndexedDB, Security, Forensic Test, Encase

Abstract

The new HTML5 standard provides much more access to client resources, such as user location and local data storage. Unfortunately, this greater access may create new security risks that potentially can yield new threats to user privacy and web attacks. One of these security risks lies with the HTML5 client-side database. It appears that data stored on the client file system is unencrypted. Therefore, any stored data might be at risk of exposure. This paper explains and performs a security investigation into how the data is stored on client local file systems. The investigation was undertaken using Firefox and Chrome web browsers, and Encase (a computer forensic tool), was used to examine the stored data. This paper describes how the data can be retrieved after an application deletes the client side database. Finally, based on our findings, we propose a solution to correct any potential issues and security risks, and recommend ways to store data securely on local file systems.

1 Introduction

While HTML5 is still in the process of being standardized by the W3C [1], its adoption will greatly help developers resolve recognized problems, such as media and online data handling; thereby providing a more robust method for handling data [10]. Furthermore, the enhanced functionalities of HTML5, such as a client-side database called IndexedDB (which will be embedded within the web browser), will provide additional benefits, such as reducing the web server load. However, while client-side databases have the advantage of reducing load on the web server, their performance will be dependent on the user's web browser; particularly, how the browser implements the new client side database API; otherwise known as the IndexedDB API.
IndexedDB is storing data on client side, which provides an offline functionality. This means that data can be accessed even without network connection. This is a big advantage when using mobile phone, which is bringing better batter life and performance.
This paper will focus on the security of this new browser-based storage capability, and a series of experiments will show how vulnerable the IndexedDB API is to attacks. These attacks will be described in more detail later, after which we will propose methods of protecting against such attacks. This paper will also investigate how the web application will store the data in the client-side database, and a series of tests will be conducted to retrieve the deleted database files. A possible solution for storing and retrieving data in secure manner will be proposed and described in further detail.
The testing will use Firefox and Chrome browsers, as they currently support the IndexedDB client-side database. The investigation will focus on the data storage mechanism of the client-side database. To help us analyse the results, a forensic tool called Encase was used; Encase is an industry standard computer forensics tool, used in the majority of criminal cases involving the collection and presentation of digital evidence [5]. Encase is a software to access raw data, and provide the functionality to create disk images.

2 Background

The development of new Web technologies faces a trade-off between stronger security (thereby protecting the user), and increased functionality (thereby helping the user). Unfortunately, this trade-off may have resulted in the development and implementation of an insecure API, namely IndexedDB API. It should be noted that the implementation of the IndexedDB into Web browsers is not yet fully completed; consequently some of these security risks may no longer exist in future implementations of the IndexedDB API. The security issue with the unencrypted data stored into IndexeddB is considerably a structure flaw. This means that the database is designed to store all of the data in unencrypted state.

2.1 Problem identification

IndexedDB is storing data in unencrypted state. This information might not be sensitive, such as usernames or password, but can include client name, address, place of birth or date of birth.
If some of the information is put together, then this can lead to identity theft.
To prevent leaking of data all over the place, we propose an algorithm to secure this information, and prevent the end user from identity theft.
IndexedDB also works on mobile devices, where the data is stored into internal phone memory. Therefore, the problem also exists on mobile device, which is more serious compared to desktop. The deleted data can be retrieved from any mobile device. As the data is unencrypted, the

security issue is much higher. Considering the scenario where the mobile phone is lost or stolen, it will be possible to retrieve the deleted data. This risk of data exposure is much higher in this case.
Compared to storing data on server, where data is not available to recover after deletion, storing on client side is going to be more insecure.

2.2 Related Work

Previous versions of HTML5 client side databases as Web SQL, LocalStorage lacks the encryption of data. The security of data stored in client side database follows the same principles as cookies, more details in section 3. On client side the only protection is firewall. Compared to storing data on server or cloud, the encryption is done separate [15]. As described, the data is stored encrypted, and will be decrypted only in user browser.

2.3 IndexedDB Structure

Files and data stored by the browser are retained on the file storage system, on the computer's hard drive. The client-side database, IndexedDB, is a persistent client-side database, consequently the files reside on the user file system and can be recovered until they are overwritten by other files.
IndexedDB treats file data just like any other type of data. An application can write a file (or Blob), into IndexedDB, as well as storing strings, numbers and JavaScript objects [6]. This is detailed in the IndexedDB specifications and, so far, implemented in both the Firefox and Chrome applications of IndexedDB. In Firefox and Chrome's IndexedDB implementation, the files are stored transparently, externally to the actual database; the performance of storing a file in IndexedDB is just as good as storing it in a filesystem. It does not bloat the database and slow down other operations. Moreover, reading from the file means that the implementation reads from an OS file; therefore, it is just as fast as a filesystem.
The Firefox IndexedDB implementation will, if it is storing the same Blob in multiple files, create only one copy. Writing further references to the same Blob just adds to an internal reference counter [9]. This is completely transparent to the web page; it writes data faster while using fewer resources.

2.3 Value in Database

Each record has a value, which could include anything that can be expressed in JavaScript, including: Boolean, number, string, date, object, array, regexp, undefined, and null. IndexedDB enables the storage of structured data, and unlike cookies and DOM Storage, IndexedDB provides features that enable to group, iterate, search, and filter JavaScript objects [6]. Each record consists of a key path and a matching value. These can be a simple type, such as string or date, or more advanced, such as JavaScript objects and arrays. It can include indexes for faster retrieval of records and can store large amount of objects.

IndexedDB is a key-value store in the same way as Local storage. However, Local storage just retains strings only key; therefore, the usual approach to local storage is to JSON.stringify it. While this is suitable for finding the object with key uniq, the only way to retrieve the properties of myObject from local storage is to JSON.parse the object and examine it. IndexedDB can store data other than strings in the value, including simple types such as DOMString and Date as well as Object and Array instances [10]. Furthermore, it can create indexes on object properties containing a specific value. So while IndexedDB can hold the same one-thousand objects, it can also create an index on the b property and use that to retrieve only the objects where b==2 without having to scan every object in the store. Furthermore, IndexedDB is aware of ranges; therefore, it can search and retrieve all records beginning with 'ab' and ending with abd' in order to find 'abc' etc.

IndexedDB is implemented differently across browsers. Firefox uses SQLite and Chrome LevelDB. It should be noted that LevelDB is not a SQL database. Like other NoSQL and Dbm stores, it does not have a relational data model, it does not support SQL queries, and it has no support for indexes.

IndexedDB is implemented in the browser on top of another database. This mean that it does not work on its own, as it is an API layer. IndexedDB is storing the value in local filesystem, which means that the limit of storage is limited to space on user hard drive. When compared to other databases, IndexedDB is updating the whole data rather that just the bits of specific data values.

3 Potential attack vector

This section is considering an unauthorized physical access attack to IndexedDB file, from outside the user local machine.

3.1 CORS (Cross-origin resource sharing) Attack

CORS is a mechanism that allows JavaScript on a web page to make XMLHttpRequests to another domain, not the domain the JavaScript originated from. Normally, web browsers would otherwise forbid such 'cross-domain' requests. CORS defines a way in which the browser and the server can interact to determine whether or not to allow the cross-origin request [12]. By letting third party applications accessing the data created with other domains application can lead to security issues, such as information leakage. Therefore user agents must implement Cross-origin resource sharing with IndexedDB in greater security details. Also, in some CORS should not be allowed, to protect the privacy of the end user.

Scenario 1: Unauthorized physical access to the OS file system, where the data from the browser database (IndexedDB) is stored unencrypted.

CORS expands on the design of the Same Origin Policy. Each resource declares a set of origins, which are able to issue various kinds of requests (such as DELETE, INSERT, UPDATE) to, and read the contents of, the resource. CORS is a "blind response" technique controlled by an extra HTTP header (origin), which, when added, allows the request to reach the target. This means, that an application creates an IndexedDB database, which is saved with the domain name. Another application cannot access the database files, as the access is restricted for the particular domain. This attack is based on bypassing the Same Origin Policy and establishing cross-domain connections to allow the deployment of a Cross-site Request Forgery attack vector [11].

Scenario 2 (Data Breach): Unauthorized access from an external machine (bypassing the Same Origin Policy (SOP)) to read the data and retrieve the information stored in the IndexedDB files.

But why is the ability to read and retrieve data stored in the IndexedDB files such an issue. In order to demonstrate the problem, we will firstly conduct an analysis of the IndexedDB database file.

4 Analysis of indexeddb database file

The first step in conducting this analysis is to build a 'clean' Hard Disk Drive (HDD) on a PC [HP Z400 6-DIMM with 12GB RAM and XEON 4 physical cores (8logical cores)] that will include the operating system (Windows 7, 64-bit) and web browsers (Firefox v.20.0.1; Chrome v.29.0.1547.620) after which some initial Internet browsing will be conducted. The HDD will then be 'acquired' using Encase v.6.11.1. This will be the starting point for each investigation. After each experiment, the image needs to be restored to a 'clean' state, and following each experiment, the disk will be forensically wiped, and then same component (operating system, web browsers) will be installed.
The aim of these experiments is to investigate and show how the data is deleted from IndexedDB (local file) and also if the data is in unencrypted state. We will also perform an reuse of recovered file and show, if it can be successfully achieved.

EXPERIMENT 1: RECOVERY OF DELETED INDEXEDDB SQLITE DATABASE FILE

In this experiment, the SQLite database file will be deleted from a Hard Disk Dive (HDD), in a PC [HP Modified to i5 processors and 16GB Ram] running a Windows 7 64-bit Operating System. Then using Encase v.6.11.1, locate the deleted data and perform a data recovery. The structure of the web browsers (Firefox v.20.0.1 and Chrome v.29.0.1547.62) will also be examined to assess how the data is stored.

EXPERIMENT 1: RESULTS

Firefox stores all data in a temporary table (SQLite database) from where the data is copied into an Object Store, complete with key/value link. After the data has been copied successfully, the temporary table is dropped. The browser always stored the SQL file in the same location in the file system. On Firefox the location is C:\Users\[user-name]\Application Data\Mozilla\Firefox\Profiles\[profile name.default]\indexedDB\[domain-name]\[database-name] where on Chrome C:\Users[user-name]\AppData\Local\Google\Chrome\User Data\Default\IndexedDB. Consequently, and previously stored data is always overwritten. Interestingly, when the data is deleted from the application (using delete function), the location within the file system is reserved for that deleted file. It is keeping the reserved location, because the deleted file still persists on the HDD. So when running the application again, the browser always allocates a different location for the newly created Object Store.

Allocation of file storage in Chrome is slightly different; all of the databases are stored in the same file. Consequently, it is assumed that Chrome is using compression for storing browsing data.

In Encase we choose the option for Copy/UnErase the deleted file. This exported the deleted file with all the data. While the deleted file data can be read from Encase, we choose to export the file and opened with SQLite Manager (Figure 1). In this way all the data in table was visible, and the field values in the `Blob` could be exported unencrypted.

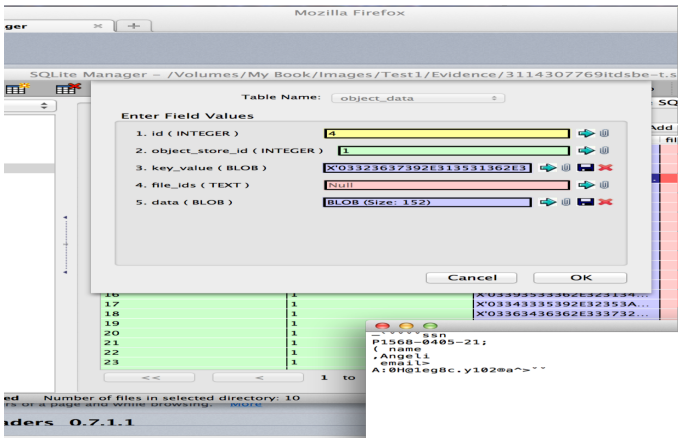

Figure 1: Exported deleted database file.

EXPERIMENT 2: CLEARING THE BROWSER CACHE

Experiments in Firefox will include deleting the data by clearing the browser cache (deleting the offline data option). Each experiment will store 300K records with a file size of 127MB. Experiments in chrome will include deleting the data by clearing the browser cache (clearing browsing data-

Hosted app date). Each experiment will store 300K records with a file size of 128MB.

EXPERIMENT 2: RESULTS

Clearing the browser cache in Chrome clears the databases and deletes the file where it is stored.
In Firefox the clearing the cache does not delete the database file from local file system.

EXPERIMENT 3: REUSE OF RECOVERED INDEXEDDB DATABASE

In this experiment the possibility of reusing a recovered IndexedDB database in a different web browser was investigated. This involved identifying the location (physical address on the HDD), of the file after it had been deleted. In addition, this experiment also considered if the database name is changed after it has been deleted; to see if the Web application can read deleted file with different filename. When deleting a database from the application, everything in the folders is deleted; including those that can be stored locally (image, document, video, audio). SQLite is not a typed database, which means that any data type can be put into any cell, regardless of the type declared for the column, and the database will attempt to convert it. Similarly, a different type than the column type is requested, SQLite will also convert this value.

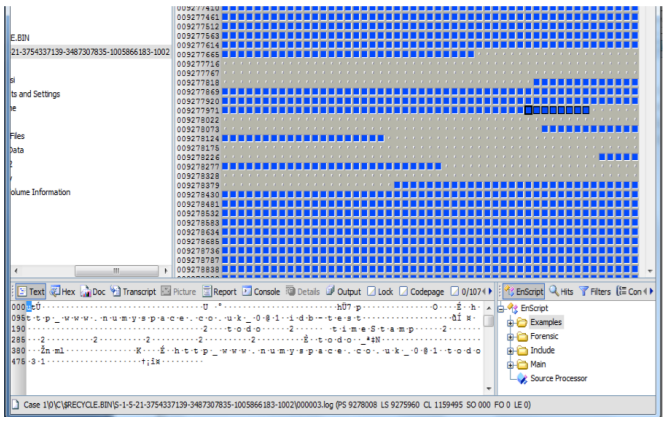

Figure 2: The physical address, and data in database file.

EXPERIMENT 3: RESULTS

Figure 2 displays the physical address of the file before and after deletion. Also the physical address of the newly created database file persists on the same location. The deleted files are marked with red cross. This file was restored with Encase and exported to another hard drive. The file was restored into database folder, and application was run to check if the data could be accessed. The result is that the application read the file and all of the data in unencrypted state was available for us to see.

5 Analyses and Possible Solution

The results were as expected; the deleted data has been marked as deleted, but it can be exported and all the information inside the database could be read. Moreover, exported data that has been imported to another PC running Windows 7 can be accessed and re-used. However a possible solution to this security issue is presented below.

5.1 A Proposed Solution to Security issue in IndexedDB

In this section we are going to propose a solution to IndexedDB storage security issue.
The prevention against such scenarios might include encryption of the files stored by the browser on the file system. All the data stored by the browser will be encrypted and stored to the file system. When retrieving the data, a secure key will be required to read the data from the file system. An encryption library will generate this key, which will permit access to read the data. Otherwise, the data remains encrypted and impossible to read. The encryption key will be downloaded dynamically and the key (password) will be stored in session key. When the key is secure, then it will encrypt data. When a user closes browser, then the key is overwritten in RAM. This will help to prevent attacker getting access to secure key when reading data from RAM.

The algorithm to secure saving of data could be a JavaScript library (proposed Stanford JS Encryption library), which will help us to prevent saving data in unencrypted state.
We going to explain steps to write, update and read the data with algorithm in pseudo code.

The following steps are described writing or updating data to database.

1. Ensure we have established the secure connection trough OAuth - The first step is to provide a secure login functionality, which can be provided by web application. The web application will use the login to authenticate a user and securely logged the user into system.
2. Open a connection to database - When an application requests a new transaction for IndexedDB to open database and save data, the designed encryption library extension will encrypt the data. This way the data will be stored in an encrypted state and not readable to others.
3. Encryption library generates public and private key - When the data is encrypted a key will be generated and stored with the user information on server. Client-side encrypts sensitive data using the public key, which will be generated and stored on sever side. This public key is used when encrypting information using the JavaScript library.

Public and private key are created simultaneously using the same algorithm (RSA- Rivest-Shamir-Adleman).

4. Encrypt data - When Client-side encryption is enabled, an RSA keypair is generated and user will be given a specially formatted version of the public key. RSA is the algorithm that is used to encrypt data with a private key to produce a digital signature [13]. The private key, however, is never revealed to user or anyone else. Once the data servers, the data is decrypted using the keypair's private key [14]. Private key is used to decrypt text that has been encrypted with public key. It uses the industry-standard AES algorithm at 128, 192 or 256 bits; the SHA256 hash function; the HMAC authentication code.
5. Save the file and close the connection

When reading the data, the following steps needs to be fulfilled.

1. Checking user credentials - When the user request the read the data from database, the web application will check user credentials (if the session is active) and get the key from server to allow decryption of data.
2. Get the key to decrypt data - Upon successful authentication user will be given a public key, which will be used for decryption of the data. This private key will be stored on server side, with all the user information, which is used for decrypt the data. We are going to use OAuth 2, which is an open standard for authorization. This will be used to securely transfer private key to server.
3. Decrypt Data – Encryption library will check for matching combination of private and public key, and perform decryption of data.
4. Show the decrypted data to user
5. Close the connection

For a secure authentication with server we are going to consider OAuth. This will provide authentication between the application and web server using a security token. We do not consider security issues with OAuth, because this will be done in later stage, when the implementation is done.
The data is stored unencrypted to file system, which can be accessed by the web application. When an application send a request to web browser to store the data on local file system, the cryptography library will encrypt the data, to be stored secure. A secure key will be also generated and stored on web server. Reading the data from local file system will be possible only when a secure key is provided and the authentication between web application and server is established. Considering all of the points are made and connection is securely established, the data is decrypted by cryptography library and displayed trough web browser to user. In fig. 3 we highlight the proposed solution showing how the cryptography library will be implemented. The library will be implemented on top of web browser API. It should be noted, that at this stage this solution is only theoretical, however further work will be undertaken to prove this hypothesis.

The algorithm will consist of the following components, which are build into browser (Figure 4).

- Mechanism for generating private and public key
- Mechanism for checking the combination of keys
- Encryption
- Decryption

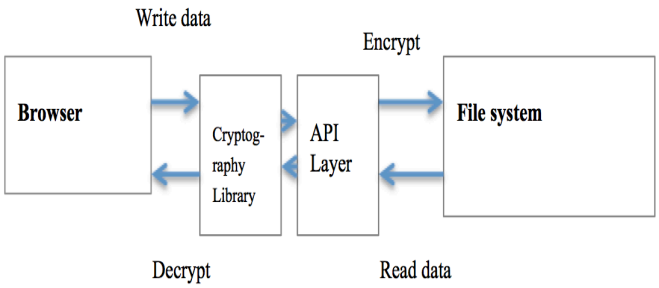

Figure 3: Proposed Encryption Library

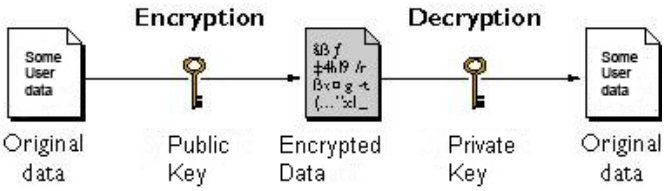

Figure 4: Encryption and decryption using keys

Encryption/Cryptography library is a piece of software or code, which encrypts readable text into unreadable data. This data can be accessed by using an encryption key. Some examples of encryption libraries are listed below. These are just few encryption libraries, which are considered for implementation into browser. This libraries were chosen, because provide the functionality to encrypt on client side, and also are available as open source.

OpenSSL: Open Source toolkit implementing the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) protocols as well as a full-strength general purpose cryptography library [4].

Crypto++: A free C++ library for cryptography: includes ciphers, message authentication codes, one-way hash functions, public-key cryptosystems and key agreement schemes [2].

GPGME: (GnuPG Made Easy) a C language library that allows support for cryptography to be added to a programme. It is designed to provide easier access to public key crypto engines like GnuPG or GpgSM. GPGME provides a high-level crypto API for encryption, decryption, signing, signature verification and key management [7].

BeeCrypt: A C++ API cryptography library [3].

Libgcrypt: GNU's basic cryptographic library [8].

Off course, another possible solution to problem might include usage of an external device to store data from a browser. For example, a user could specify to a location to which any IndexedDB files should be stored when browsing the web or running some applications. This includes an option where the data could be written and read from an external source, such as USB. The USB key will need to be secured with access encryption and restricted to access data when the master password is entered.

6 Conclusion

In this paper, we have demonstrated security related flaws within IndexedDB. While the browser can delete IndexedDB files stored on the local filesystem, they can be retrieved by Encase. Unfortunately, the retrieved data is in an unencrypted format, and given the nature of the data held within the IndexedDB API, a potential security issue exists.
All the data in IndexedDB is exposed. We have demonstrated a solution for this security issue, which includes a secure 'library', located between the browser and the filesystem. All data stored by the Indexed DB application will be encrypted and saved to the library. Therefore, if the application needs to read the data, an encryption key will be required. Without a key, the data will not be decrypted and reading the data will not be possible. This will help to secure the data stored on the client side and prevent retrieval in an unencrypted state. Future work will focus on implementing the cryptography library into web browser and testing for possible attacks.

References

[1] Berjon, R. (2014) W3C HTML5 Specification. Available at: http://www.w3.org/html/wg/drafts/html/master/

[2] Dai, W. (2004) Cryptoo++ Library. Availabe at: http://www.cryptopp.com Last Accessed :20 May 2014

[3] Doxygen (2009) BeeCrypt C++ API Documentation. Availabe at: http://beecrypt.sourceforge.net/doxygen/c++/index.html Last Accessed :20 May 2014

[4] Engelschall, R. S. (1999). About the OpenSSL Project Available at: https://www.openssl.org/about/

[5] Simmons, M.Chi, H. (2012) Designing and implementing cloud-based digital forensics hands-on labs. Proceeding InfoSecCD '12 Proceedings of the 2012 Information Security Curriculum Development Conference. Pages 69-74

[6] Flanagan, D. (2011) JavaScript: The Definitive Guide Activate Your Web Pages. 6th edition, Publisher: O'Reilly

[7] Koch, W. (1999) GPGME – The GNU Privacy Guard. Available at: https://www.gnupg.org/index.html Last Accesed : 20 May 2014

[8] Koch, W. (2003) Libgcrypt. Available at: http://www.gnu.org/software/libgcrypt/ Last Accesed : 20 May 2014

[9] Mehta, N. Sicking, J. Graff, E. Popescu, A. Orlow, J. Bell, J. (2013) Indexed Database API. Available at: http://www.w3.org/TR/IndexedDB/ Accessed on: 5th July 2014

[10] Sarris, S. (2013) HTML5 Unleashed. Published by: Sams. Print ISBN-10**:** 0-672-33627-8

[11] Stuttard, D. Pinco, M. (2007) *Web Application Hacker's Handbook*. Published by: Wiley Publishing, Indianapolis, Indiana

[12] Zakas, C. N. (2010) Cross-domain Ajax with Cross-Origin Resource Sharing". NCZOnline. Available at: http://www.nczonline.net/blog/2010/05/25/cross-domain-ajax-with-cross-origin-resource-sharing. Last Accessed: 20 February 2014

[13] Bernett, S. Paine, S. (2001) RSA Security's Official Guide toCryptography. Osborne/McGraw-Hill

[14] Mollin, R. (2003) RSA and Public-Key Cryptography. CHAPMAN & HALL/CRC A CRC Press Company ISBN 1-58488-338-3

[15] Popa, A.D. Stark, E. Valdez, S. Zeldovich, N.Kaashoek, F.M. Balakrishnan, H. (2014) Building web applications on top of encrypted data using Mylar. 11th USENIX Conference on Networked System Design and Implementation. Pages 157-172